Programming Assignment Report

Program 4

CPU scheduling

Abstract

The purpose of this lab was to learn and understand how to use Round Robin and First Come First Served CPU scheduling in Java. In the lab, a CPU scheduler was implemented to manage multiple processes and compute key performance metrics such as CPU Utilization, Throughput, Average Waiting Time, and Average Turnaround Time. The output from both algorithms was compared to evaluate their strengths and weaknesses in terms of fairness, efficiency, and responsiveness. Through this lab, the differences in how FCFS and RR handle process execution was shown.

Introduction

3.1 Problem Statement

Understand how FCFS and RR work and program in java using the provided code as a base.

3.2 Objective

To understand how different CPU schedulers can have an effect on the runtime of different processes.

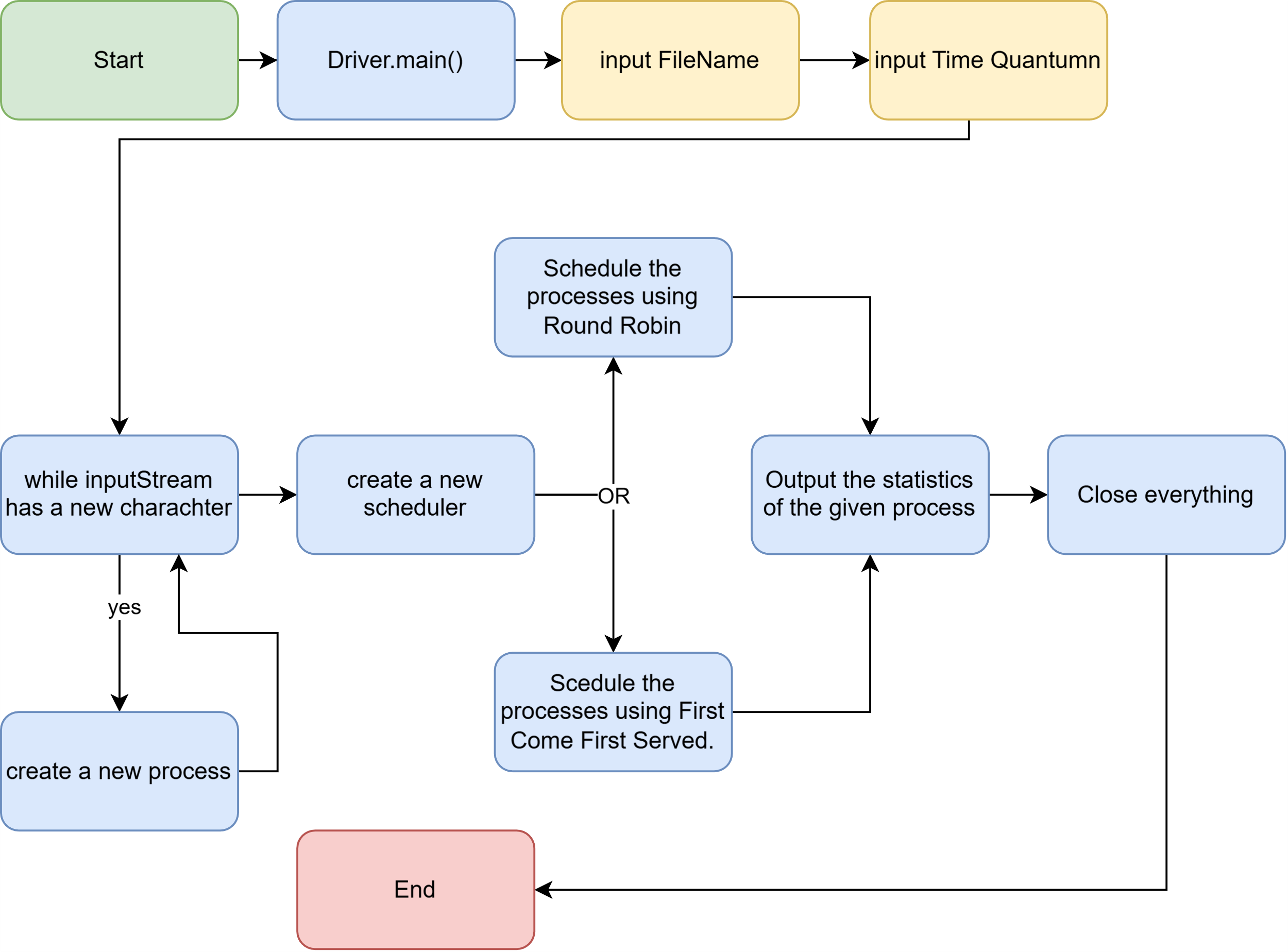
3.3 Background Information

The concepts used within the lab relate to CPU Scheduling, more specifically First Come First Served and Round Robin.

Materials and Methods

4.1 Tools and Technologies

For the assignment we used several different software types to help complete the lab. For the main part of the assignment, we used a Virtual ubuntu machine to run our Linux environment. Within the environment we used the terminal to access all the operating systems programs. We used the nano text editor to write Java code to then compile using javac compiler. We then were given multiple starting files that had missing portions to fill out. Once filled out we would run it with each separate scheduling method by uncommenting the specific line.

4.2 Program Design  


4.3 Code Implementation

Using the provided files I implemented the missing code for statistic calculations, round robin implementation, and first come first served. When testing if it worked, I verified that it first compiled correctly, then tested using the example input and the input csv file.

Results

5.1 Test Cases

I ran each of the given scripts to test the outputs of each file. The outputs are below.

5.2 Output

Round Robin

A screenshot of a computer program

AI-generated content may be incorrect.

First Come First Serve

A screenshot of a computer program

AI-generated content may be incorrect.

Discussion

6.1 Post-Lab Questions

N/A

6.2 Challenges Encountered and Improvements

A challenge I faced when implementing the code was the fact that response time was not properly added to cpu. I first had to add it to the process Class and then add it to the schedule class for each scheduling type.

Conclusion

I understand the pros and cons for Round Robin and First come first served. Round robin is faster in response time but slower in waiting time and turn around time. First come first served is faster in both turn around time and waiting time.

Learned

|  |  |  |
| --- | --- | --- |
| Pro/cons | RR | FCFS |
| PRO | Faster Average Response Time | Faster Average Waiting time |
| PRO |  | Faster average turnaround time |
| CON | Slower Average Waiting Time | Slower average response time |
| CON | Slower average turnaround time |  |

Source Code:

Racer.java

|  |
| --- |
| Scheduler.java  import java.util.\*;  import java.util.ArrayList;  *// Source code*  public class Scheduler {      int timer;      ArrayList<Process> listOfProcesses;      ArrayList<Process> readyQueue;      ArrayList<Process> endProcesses;      int timeQuantum;      int contextSwitch; *// small number (less than half the timeQuantum).. ex:*  *// 0.1 \* timeQuantum*      Process cpu;      int counter;      public Scheduler(ArrayList<Process> listOfProcesses, int timeQuantum) {          this.timeQuantum = timeQuantum;          this.listOfProcesses = listOfProcesses;      }      public void fcfs() {          timer = 0;          contextSwitch = 0;          cpu = null;          readyQueue = new ArrayList<>();          endProcesses = new ArrayList<>();  */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  *\* Fill the code here (pseudo code is below)*  *\* 1. Sort the processes according to arrivalTime*  *\* 2. For each process*  *\* set process's responseTime*  *\* update timer according to the burstTime*  *\* set process's completionTime*  *\* add it to endProcesses*  *\* 3. Update contextSwitch by 1 when the process is completed*  *\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*  *// 1. Sort the processes according to arrivalTime*          Collections.sort(listOfProcesses, Comparator.comparingInt(p -> p.arrivalTime));  *// 2. For each process*          for (int i = 0; i < listOfProcesses.size(); i++) {              Process currentProcess = listOfProcesses.get(i);  *// set process's responseTime*              currentProcess.responseTime = timer-currentProcess.arrivalTime;  *// update timer according to the burstTime*              timer += currentProcess.burstTime;  *// set process's completionTime*              currentProcess.completionTime = timer;  *// add it to endProcesses*              endProcesses.add(currentProcess);          }  *// 3. Update contextSwitch by 1 when the process is completed*          contextSwitch = endProcesses.size();          System.out.println("-----------------FCFS----------------");          Utilities myUtility = new Utilities(endProcesses, contextSwitch, timer);          myUtility.calUtilities();      }      public void rr() {          timer = 0;          contextSwitch = 0;          cpu = null;          readyQueue = new ArrayList<>();          endProcesses = new ArrayList<>();          while (!readyQueue.isEmpty() || !listOfProcesses.isEmpty() || cpu != null) {  *// add to readyQueue*              for (int i = 0; i < listOfProcesses.size(); i++) {                  if (listOfProcesses.get(i).arrivalTime == timer) {                      readyQueue.add(listOfProcesses.remove(i));                  }              }  *// add to cpu*              if (cpu == null) {                  cpu = readyQueue.remove(0);              }              counter++;              cpu.serviceTime++;  */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  *\* Fill the code here (pseudo code is below)*  *\* 1. If this process is new to be allocated to cpu, set its responseTime*  *\* 2. If the process is completed by comparing cpu.serviceTime == cpu.burstTime,*  *\* set process's completionTime*  *\* add it to endProcesses*  *\* clear up cpu (cpu = null)*  *\* increase contextSwitch by 1*  *\* reset counter to 0*  *\* 2. else if the process uses up the timeQuantum*  *\* put it back to readyQueue*  *\* clear cpu (cpu = null)*  *\* increase contextSwitch by 1*  *\* reset counter to 0*  *\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*  *// 1. If this process is new to be allocated to cpu, set its responseTime*              if (cpu.serviceTime == 1) {                  cpu.responseTime = timer-cpu.arrivalTime;              }  *// 2. If the process is completed by comparing cpu.serviceTime == cpu.burstTime,*              if (cpu.serviceTime == cpu.burstTime) {  *// set process's completionTime*                  cpu.completionTime = timer;  *// add it to endProcesses*                  endProcesses.add(cpu);  *// clear up cpu (cpu = null)*                  cpu = null;  *// increase contextSwitch by 1*                  contextSwitch++;  *// reset counter to 0*                  counter = 0;              }  *// 2. else if the process uses up the timeQuantum*              else if (counter == timeQuantum) {  *// put it back to readyQueue*                  readyQueue.add(cpu);  *// clear cpu (cpu = null)*                  cpu = null;  *// increase contextSwitch by 1*                  contextSwitch++;  *// reset counter to 0*                  counter = 0;              }              timer++; *// real time : cpu time*          }          System.out.println("-----------------RR------------------");          Utilities myUtility = new Utilities(endProcesses, contextSwitch, timer);          myUtility.calUtilities();      }  }  Utilities.java *// Source code is decompiled from a .class file using FernFlower decompiler.*  import java.util.ArrayList;  public class Utilities {     ArrayList<Process> endProcesses;     int contextSwitch;     int timer;     public Utilities(ArrayList<Process> var1, int var2, int var3) {        this.endProcesses = var1;        this.contextSwitch = var2;        this.timer = var3;     }     public void calUtilities() {        double var1 = 0.0;        double var3 = 0.0;        double var5 = 0.0;        double var7 = 0.0;        for(int var9 = 0; var9 < this.endProcesses.size(); ++var9) {           Process var10 = (Process)this.endProcesses.get(var9);           var1 += (double)(var10.completionTime - var10.arrivalTime);           var3 += (double)(var10.completionTime - var10.arrivalTime - var10.burstTime);           var5 += (double)var10.burstTime;           var7 += (double)var10.serviceTime;        }        double var19 = var5 / (var7 + var1);        double var11 = (double)(this.endProcesses.size() / this.timer);        double var13 = var7 / (double)this.endProcesses.size();        double var15 = var3 / (double)this.endProcesses.size();        double var17 = var1 / (double)this.endProcesses.size();        System.out.println("");        System.out.println("CPU Utilization: " + var19);        System.out.println("Throughput: " + var11);        System.out.println("Average Response Time: " + var13);        System.out.println("Average Waiting Time: " + var15);        System.out.println("Average Turnaround Time: " + var17);     }  } |

Appendix

n/a